Spring 4-1-1990

Mapping Data to Processors in Distributed Memory Computations ; CU-CS-469-90

Matthew Rosing
University of Colorado Boulder

Robert P. Weaver
University of Colorado Boulder

Mapping Data to Processors in Distributed Memory Computations

Matthew Rosing and Robert P. Weaver

CU-CS-469-90 April 1990

Department of Computer Science
Campus Box 430
University of Colorado,
Boulder, Colorado, 80309 USA

This research was supported by AFOSR grant AFOSR-85-0251, and NSF Cooperative Agreement CDA-8420944.

To appear in Proceedings of the Fifth Distributed Memory Computing Conference, held in Charleston, South Carolina, April 1990

Mapping Data to Processors in Distributed Memory Computations

Matthew Rosing
Robert P. Weaver

Department of Computer Science
University of Colorado
Boulder, Colorado 80309

Abstract

We present a structured scheme for allowing a programmer to specify the mapping of data to distributed memory multiprocessors. This scheme lets the programmer specify information about communication patterns as well as information about distributing data structures onto processors (including partitioning with replication).

This mapping scheme allows the user to map arrays of data to arrays of processors. The user specifies how each axis of the data structure is mapped onto an axis of the processor structure. This mapping may either be one to one or one to many depending on the parallelism, load balancing, and communication requirements.

We discuss the basics of how this scheme is implemented in the DINO language, those areas in which it has worked well, the few areas where we had significant problems, and some ideas for future improvements.

1. Introduction

A crucial issue in using distributed memory multiprocessors efficiently for data parallel computation is the mapping of the data structures to the processors. By data parallel, we mean the programming paradigm in which parallelism is obtained by applying a single function to sections of a data structure concurrently.

There are two polar approaches to handling the mapping process. The compiler may examine the program and determine a mapping most suitable for the problem. Or the programmer may provide some specification of the mapping. In our approach, the programmer provides high level mapping information, and the compiler uses this information to make decisions about how to implement the particular program.

Our mapping specifications contain two general categories of information. First, the mapping specifies how data structures are distributed onto the individual processors (we use the term distributed to mean not only partitioning but also to include replication when a piece of data is needed on more than one processor). Second, the mapping may specify the communication pattern between the processors using that data structure. In this context, we think of communication as a case of replication in which the value of the replicated data will need to be updated. For a further discussion of this point, see Section 3.

We have been involved in the development of a language, DINO, for which we designed this approach. DINO requires the programmer to furnish high level mapping information for the distributed data structures with respect to a user defined structure of processors. In this paper, we draw on this experience to describe a structured approach to the design of programmer specified mappings of data to processors and to comment on some of the interesting issues raised by the design and implementation of this approach.

Any language that uses the data parallel paradigm for distributed memory programming will deal with this issue in one form or another. The mapping can be done dynamically at runtime. Linda allows the user to distribute data by distributing its tuple space across the processors, but can adjust the mappings automatically at runtime[2]. Spot's compiler does this sort of analysis automatically on invocation of the program[5]. Other languages have used programmer supplied mapping primitives[1] and at least one has provided mappings from regular data structures to regular structures of processors, the Kali Project[3]. Neither Spot nor the Kali Project has programmer

specified mappings that allow distribution with replication or that allow specification of communication patterns.

The remainder of this paper will provide a brief overview of DINO to supply a context for the following discussion, discuss some high level considerations of programmer defined mappings, present our design and implementation of mappings, discuss some of the issues and solutions raised by this implementation, and indicate some directions for future work.

2. An Overview of DINO

To provide a framework for this discussion of mapping distributed data structures, we will first give a brief overview of DINO. DINO is a language consisting of high level parallel extensions to C which is intended for writing numerical programs on distributed memory multi-processors. DINO provides the programmer with a consistent global view of the data structures even though they may have been broken up and distributed to processors.

DINO provides this support for data parallel programming in the following way: First, the programmer specifies a regular structure (an array with one or more dimensions) of virtual processors that reflects the way the problem is most naturally decomposed. For example, the programmer might specify a vector of P processors as a structure of processors. Second, for each major data structure in the problem, the programmer specifies the manner in which the data structure is mapped onto the structure of processors. This mapping is the key to DINO and the main focus of this paper. Finally, the programmer invokes a "composite procedure" that executes concurrently on each processor in the structure, using that portion of the data structure that is mapped there. This results in a SPMD (single program, multiple data) form of parallelism.

DINO provides inter processor communications in two ways. First, inter processor communication arises automatically when a distributed data structure is used as a parameter to a composite procedure. The parameter is distributed and collected automatically. Second, the programmer can designate an element of a data structure which has been mapped to more than one processor to be sent or received. This is done by using the data name followed by "#" on the left or right hand side of an assignment statement respectively. DINO then handles much of the underlying communications automatically. For more information on DINO, see [4].

There are two DINO features that make the implementation of the mapping process particularly complex. The first is that DINO provides the programmer with a global view of the data structures at all times. That is, the programmer uses the global name to access the local portion of a data structure on a given processor. The compiler must translate these accesses. Second, for efficient communications DINO allows the programmer to use "ranges" of data structures (sub-matrices). Since a range can be defined by a data dependent expression, some portion of the analysis required for communication may have to be be done at run time.

3. Our Philosophy on Mappings

We have said that we see two general categories of information that programmer specified mappings may include, information about distribution and information about communication patterns. That is, a mapping may serve two purposes. It allows the compiler to statically map the data to the processors. It may also may allow the compiler to do a better static analysis of communications resulting in more efficient communication.

The obvious purpose for a mapping is to allow the compiler to partition the data among the processors. While this may seem to simply be a matter of dividing a data structure up into "pieces" and sending each "piece" to a processor, in our approach it is somewhat more complex than this. It is often useful to have the same data value available to many processors. Although this can be implemented by having the programmer communicate the needed value once the data has been partitioned, it may be more efficient and easier for the programmer to think about the problem if the mapping allows for replication of the same "piece" on many processors.

Related to the concept of replicating data so that more than one processor may make use of it is the fact that the programmer may wish to update that value during the program. For this purpose, our mappings may also be used to provide some high level information about communication patterns. In order to have a communication, the compiler must know "what" piece of data is being communicated, "when" is it being communicated, "where" is it being communicated to (from), and "how" it is communicated. Our approach allows the programmer to specify what data piece is to be communicated and when (lexically) it is to sent and received. The compiler can then determine, based on the mapping, where it is to go to (or come from) and how the communication will be

handled (all the underlying detail of messages).

To do this, we needed a communication paradigm than can easily be described in the mapping. We use the following paradigm: First, when a data structure is mapped, a partition is implicitly defined. That is, each "piece" of data is assigned to one processor that "owns" the data. We call this the "home" processor for that piece of data. Second, any other processor that will need to use that piece of data is assigned the same piece, but this processor is designated as a "copy" processor for that piece of data. Then, all sends and receives for this piece of data are from the home processor to all its copy processors. With this paradigm a parallel program is guaranteed to be deterministic, and communication patterns can be described simply by specifying the copy processors, if any, for each piece of data. (In DINO, this default paradigm can be overridden by the programmer but it is sufficient for many numerical algorithms.)

Using these high level concepts, we constructed a mapping strategy. To be useful for a large class of algorithms, we believe that this strategy should satisfy five goals. First, it should provide a wide variety of mappings so that there are natural ways to decompose data structures for most problems. Second, it should allow data to be replicated as well as simply broken-up (partitioned) across the processors. Third, it should provide the compiler with information to automate as much of the inter-processor communication as possible. Fourth, it should run efficiently at compile and run time. And fifth, it should be easy for the programmer to generate these specifications. One of the most interesting problems is how to resolve the conflicts that inevitably seem to arise between the flexibility of the mappings and the resulting efficiency and ease of use.

4. The Mapping Constructs

We have attempted to provide a general mapping mechanism that still allows efficient implementation at compile time. Our mapping specification describes how to map arrays of data onto arrays of processors and is based on describing how each axis of the data structure is mapped to the processor structure (a subspeci"014cation). By having an (almost) orthogonal set of sub-specifications, a powerful mapping specification can be constructed out of a simple set of primitives.

A mapping specification is defined in three steps. First, the programmer specifies how each axis of the data structure is matched to one or more axes of the structure of processors. Second, the programmer specifies how the data on that axis is distributed among the processors defined in the first step. There are three basic choices for this step, complete replication, partition, or partition with copies. Third, if partition with copies is desired, the programmer specifies how the copies of data are distributed to other processors.

In the first step, matching axes of the data structure to axes of the structure of processors, there are only two primitives — "compress" and "align". With compress, the programmer specifies that this data axis will not be distributed. This primitive is used if the data structure has more axes than the structure of processors. With align, the programmer specifies that a particular axis in the data structure should be mapped to a particular axis in structure of processors. If this primitive is omitted for an axis in the data structure, the obvious default of mapping the next available data axis to the next processor axis is used. (A data axis might not be available because it has been designated as "compress".)

For example, if the programmer is mapping a two dimensional matrix to a vector of processors, a mapping specification of the form:

```
[ . . . ][compress]
```

will distribute rows to the processors and a specification of the form:

```
[compress][ . . . ]
```

will distribute columns. The specific mapping of the rows or columns will depend on what goes in the `[ . . . ]` sub-specification, something that the programmer determines in steps two and three.

If the programmer is mapping a two dimensional matrix to a two dimensional structure of processors, the mapping specification:

```
[ . . .  align 1][ . . .  align 0]
```

would cause the second axis of the data structure to map to the first axis of the structure of processors, and the first axis of the data structure to map to the second axis of the structure of processors. In effect this allows the matrix to be transposed and placed on the processors.

In the second step, the distribution of a data axis to a processor axis, there are three mapping primitives — "all", "block", and "wrap". Using the first of these, the programmer can specify that the data axis be completely replicated across the associated

axis of the structure of processors. With the second or third, the programmer can specify that the data axis is distributed in either one of two ways — blocked or wrapped. Block mappings assign one (approximately) equal sized contiguous piece of the data structure to each processor. Wrap mappings assign every Pth position on the data axis to the same processor (assuming P processors). Wrap is essentially a variant on block that is used for improved load balancing in a wide variety of parallel numerical algorithms. The programmer may also specify the width of the wrap.

If, in our first example, the matrix is N by N and there are P processors with $N = 4P$, then a mapping specification with:

```
[block] [compress]
```

would distribute four contiguous rows to each processor. Alternatively,

```
[compress] [wrap]
```

would put column 0, P, $2P$, ...on processor 0, etc.

In the third step, the specification of a distribution for copies, there are two mapping primitives that the programmer can use — "overlap", and "cross". "overlap" specifies that copies of neighboring data points on that axis will be available on each processor. The programmer can specify the direction and depth of the overlap. "cross" allows the programmer to specify that copies will be available for data points that are found in the intersection of two or more overlaps — in effect making copies of neighboring data points along diagonals available on each processor. This might be used, for example, in an algorithm that requires a nine point stencil.

For example, if a vector $[x_0 \quad x_1 \quad x_2 \quad x_3]$ is distributed across a vector of four processors with the mapping specification

```
[block overlap 1,1]
```

the four processors will have the following elements:

processor 0	$[x_0$	x_1		$]$
processor 1	$[x_0$	x_1	x_2	$]$
processor 2	$[$	x_1	x_2	$x_3]$
processor 3	$[$		x_2	$x_3]$

The notation "`1,1`" specifies overlaps of one element to the left and right respectively. Note that the "leftmost" and the "rightmost" processors will receive 1 element that is a copy instead of two, i.e., there is no wrap around. The home processor of each data element is the processor it would be mapped to if the overlap were omitted, in this case processor i for x_i.

If we distribute an N by N matrix across an N by N structure of processors with the following mapping specification:

```
[block overlap 1,1 cross 1]
        [block overlap 1, 1]
```

then the data on each processor (except the edge processors) will have the following pattern (nine point stencil):

$$\begin{bmatrix} c & N & c \\ W & D & E \\ c & S & c \end{bmatrix}$$

where D is the data for that processor, N, S, E, and W are copies of data from four neighboring processors due to the overlap primitive, and the c's are copies of data from the four diagonal neighbors due to the cross primitive.

Using this structured approach allows us to combine a rather small set of primitives for each data axis to generate a very large set of mappings from data structures to processors. For example, if an N by N matrix is to be distributed across a vector of N processors so that each processor received a column plus copies of the two columns to the left of "its" column, the mapping would be:

```
[compress] [block overlap 2,0]
```

The "leftmost" and the "next leftmost" processors will receive, respectively, 0 and 1 columns that are copies.

As a final example, if an N by N matrix is to be distributed across a vector of P processors, where N is 16 and P is 4, so that each processor receives copies of the closest row from the two processors next to it, the mapping would be:

```
[block overlap 1,1] [compress]
```

The resulting data structure on a processor (for an interior processor) would be:

$$\begin{bmatrix} u & u & u & u & u & u & u & u \\ D & D & D & D & D & D & D & D \\ D & D & D & D & D & D & D & D \\ D & D & D & D & D & D & D & D \\ D & D & D & D & D & D & D & D \\ d & d & d & d & d & d & d & d \end{bmatrix}$$

where D is the home data for that processor, and u and d are copies of data from the processors above and below respectively.

5. Implementation and Performance

In this section we will discuss the results of incorporating this mapping strategy into DINO and how our implementation has affected the efficiency of the resulting DINO programs. We will examine the results that flow from our choice of a structured mapping strategy as well as the results of the implementation methods we chose.

When we first laid out our mapping strategy, we attempted to balance the flexibility of the mappings (the potential number of useful mappings we could construct) against the efficiency of the implementation. For example, we rejected a completely random mapping strategy because it appeared to be too complex to implement (in addition, it was not clear how a programmer would specify such a mapping for large data structures). Instead, we elected to follow a strategy that (for the most part) treated each dimension of the data structure as orthogonal to every other dimension and only considered mappings where the mapping in a given dimension could be set out with a simple expression. For the most part, this appears to have been a good choice. The one problem we found is due to the use of the wrap primitive.

To make this discussion understandable, we need to provide some high level overview of and motivation for the implementation of DINO. Essentially, DINO is composed of two parts, a compiler and a run time library. Together, these parts perform two basic functions that are related to the mappings. They translate programmer accesses to data structures that have been distributed into the actual accesses to the local data structures and they implement the communications. These concepts are discussed in some more detail in Section. 5.1.

When we designed the compiler and the library for DINO, we had to chose between putting the burden of doing the necessary analysis on the compiler or on the library, and we had to chose between implementing each part of the analysis as a single large general purpose module or implementing each part as a series of specialized modules. In both cases, we did some of each. We had mostly good results with this approach but there have been a few problems.

Initially we decided to do all of the access translation analysis in the compiler and to do all of the analysis required for communication in the library. We felt that the access translation was more critical to efficiency than the communication analysis. In addition, the analysis associated with the communication is much more complex than the access translation. It was easier to design a single set of library modules to handle the general communication case than it was to determine how much the compiler could do versus what had to be done at run time.

For the same reasons, the compiler generates the expressions necessary for access translation in such a way that they are tailored to the specific data structure, processor structure, and mapping. But we took the opposite approach with communications. Partly because the communications analysis is much more complex, we decided to solve the general problem first, then look at specific optimizations later.

In our preliminary testing of the first implementation of DINO, all of the noticeable performance problems appear to flow principally from these three decisions. The first, the use of the wrap primitive, is discussed in Section 5.2.1. The second, our choice of a particular division of the analysis task between the compiler and the run time library, is discussed in Section 5.2.2. The third, the generality of the communications analysis, is addressed in Section 5.2.3.

Partly as a result of these problems, we added a few optimizations to the completed compiler and run time library. Overall, the results from these optimizations have been encouraging. In certain programs with simple mapping functions, the resulting execution times are within a few percent of hand coded programs. On the other hand, there are a few mappings that, in the current version, result in inefficient code. We present more information on this in Section 5.3.

5.1 The DINO Implementation

DINO's compiler does a complete syntatic and semantic analysis of the programmer's code, and if there are no mistakes, generates C code for the target machine. Some of this C code is in the form of tables that provide information to the run time library. The library is a group of support functions called by the code generated by the compiler, and happens in the first version of DINO to be mainly used for communications. No matter how sophisticated we make the compiler, some analysis will have to be done at run time because not all the information necessary to translate the DINO constructs is known until run time.

The mappings are used by the compiler and the library to translate programmer accesses to distributed data into the correct physical accesses and to generate low level communication calls that implement communication in DINO (this applies to parameter distribution and collection as well as more explicit

communication between processors). To provide a context for the rest of the discussion, we will give a brief description of each of these functions.

First, DINO uses the mappings to provide for correct programmer access to data that has been distributed to processors. Since the programmer sees the data globally but any particular processor only has enough storage for the data that is resident there, DINO must translate this difference in viewpoint. Essentially, the compiler generates an expression for each dimension in the data structure into which the programmer's access expression can be inserted and then the entire expression is evaluated for the correct access for this processor. For example, if $A[12]$ is blocked across a vector of 4 processors, and the programmer accesses $A[I]$, the "I" is translated into "$(I) - offset$" where "$offset$" is a value dependent on which processor is executing.

There are three different times at which this computation is done, compile time, beginning of runtime, and access time. In the example above, the expression "$(I) - offset$" is computed at compile time, the value of "$offset$" is determined when the procedure starts to execute on that processor and the value of "$(I) - offset$" is computed at each access. An optimizing compiler that does common subexpression elimination can reduce the amount of access time computation, especially for accesses occurring within loops. Using the same basic methodology, the compiler could generate expressions that would test the legality of accesses. This would be useful (presumably as a compiler option) in debugging, but is not done currently.

Second, DINO uses the mappings to provide for correct communications of data that is distributed. We will describe how sends of distributed data work. Receives of distributed data and parameter distribution and collection are similar. In the send case, the programmer indicates that some element or range (sub-array) of data is to be sent. DINO determines which processors are the recipients (there can be more than one), and for each processor, it generates information that is used to copy the data to be sent to that processor into a communication buffer which is then transmitted.

Because the range that the programmer specifies (which may not be known until run time) affects not only the specific piece(s) of data to be sent, but also the processors it (they) can be sent to, this communications analysis can be quite complex. For example, in our overlapped block row example (the last example in Section 4), if the programmer specifies that the entire home piece is to be sent, there are generally two other processors that it will be sent to. If, however, the programmer specifies some subset of the home piece, there may be zero, one, or two other processors involved.

Thus the compiler and the runtime library together convert DINO constructs into C code that will run on the target machine. How much of that conversion each part should do and the manner in which it should be done are addressed next.

5.2 Performance of the Original Design

Initially we will discuss the results of implementing the original design. To the extent we can, we examine the impact of using the structured mapping strategy, placing certain parts of the analysis in the library, and using general purpose modules. In the sections that follow this, we present solutions to some of the problems we identify here.

The access translation of structured mappings has turned out to be very efficient. In our preliminary timings, we have found no significant time differences between DINO programs with large number of accesses to distributed data and hand coded programs executing the same algorithm. We believe that we have shown that one can reasonably implement a distributed language where the programmer maintains a global view of data, even after it is distributeded.

For communications using structured mappings, the results are mixed. For certain simple mappings of large data structures, the efficiency of DINO when doing parameter distribution or sends and receives comes close to hand coded programs. For much smaller data structures or short messages, DINO communications are somewhat slower. In a worse case test, zero length DINO messages take two to three times longer on the Intel iPSC2 than hand coded ones. These problems are largely due to the amount of run time analysis and the use of general purpose modules and are discussed in Sections 5.2.2 and 5.2.3..

However, one communication problem was directly due to the mapping strategy. We turn to that problem first.

5.2.1 The Use of Structured Mappings. One of the mappings we chose to implement – the wrap – turned out to be fairly inefficient in practice. Our experience with this leads us to believe that certain kinds of mappings will generally present us with performance concerns. Those mappings are characterized by the fact that they put logically disjoint pieces of data on a processor. By logically disjoint, we mean that if you

draw a picture of the global data structure, all the data going to a processor is not contiguous.

To illustrate this, contrast the parameter distribution of an N by N matrix onto a vector of P processors with $N = 1024$ and $P = 16$, for the two cases of `[compress] [block]` (blocks of columns) and `[compress] [wrap]` (wrapped columns). In the block case, DINO does 1024 high speed memory to memory copies to set the data up for distribution for each processor. (We use optimized assembly language routines to do the copying.) In the wrap case, DINO does 65536 such copies for each processor — the time for copying is $O(N^2/P)$ instead of $O(N)$. To make matters worse, in the wrap case the size of each piece being copied is so small that the analysis of where the next piece starts begins to take more time than the copying. The times for the parameter distribution for these two cases on the Intel iPSC2 are 8 seconds and 177 seconds respectively.

Note, that we can only attack the second half of the problem by changing the implementation (putting most of the analysis back into the compiler). The first half remains. Solutions are discussed in Sections 5.3.1 and 5.4.1.

It is also important to understand that this problem only occurs when DINO is distributing parameters or collecting them. Once the wrap parameters are distributed to the processors, the data is no longer logically disjoint and this problem disappears.

5.2.2 Run Time Analysis. We chose to do as much of the necessary analysis for access translation as we could in the compiler. Conversely, we chose to do most of the communications analysis in the library.

Because the access translation appears to achieve good efficiency, letting the compiler do as much of the work as possible appears to have been the correct strategy.

However, the penalty we incurred in some communications is significant. For very short messages, the amount of time consumed by the preliminary analysis that must precede any send or receive is substantial in comparison to the time for the communication itself. In programs with fine grained communication, the DINO user pays a noticeable penalty over the hand coded program.

For instance, in the block overlap example (the last example given in Section 4) for a send, the run time system must first determine that there are at most two processors that might be targets (unless this is an edge processor) and which ones they are, then it must determine if data actually goes to those processors, finally it has to generate the information necessary to copy the data for each. If the the compiler were doing some of the analysis for this mapping, it could determine in advance that, at most, only two processors are involved and it could generate a simple expression to decide which one(s). We discuss this improvement in Section 5.3.1.

5.2.3 General Purpose Modules. We also chose to do most of the communications analysis in a single, general purpose module instead of using many, tailored modules. Unfortunately, it turned out that this strategy appears to penalize the simple cases by requiring a complex analysis.

For instance, in our column wrap example above, the library may have to allow for the size and shape of the data structures that make up the actual and the formal parameters (the two do not have to be the same), the size of the element, the width of the wrap (this can be greater than one), and the number of processors. All of this must be examined for a general analysis, even though in most cases the actual analysis could be much simpler.

We address the possibility of reducing the amount of this analysis in Section 5.3.1.

5.3 Currently Implemented Improvements

We now discuss two improvements to the implementation approach described above that are in the current version of DINO and that result in better performance. We first address the performance problem with certain communications discussed in Sections 5.2.2 and 5.2.3, then talk about an optimization that addresses a problem we haven't yet discussed, the message typing problem.

5.3.1 Communication improvements. The obvious alternative to the current way of handling the communications is for the compiler to generate functions or expressions to handle the analysis and copying necessary for communication that are specific to each data structure. These functions or expressions could be optimized for the specific combination of data structure, processor structure, and mapping. This step would clearly alleviate many of the communications problems we found.

It turns out that we can go one step further: the compiler can generate functions that are also tailored to the specific data access pattern involved in the particular communication. For example, if the mapping results in the block overlap example, and the compiler just knows about the data structure, the processor structure, and the mapping, then it only knows that

no more than two other processors can be involved. The programmer could specify a complex send that would involve one or both of these processors and many different combinations of the data. But if the compiler examines the programmer's specification of the send, it may be able to deduce much more than this.

For example, if the programmer writes:

```
A[id * N / P][]# = . . .
```

(where *id* is a DINO constant that has the processor index as its value and [] signifies that the whole row is involved), the compiler can determine exactly which piece of data is involved (and that it is physically contiguous) and which processor it is to be sent to.

Thus if the expression needed for the analysis of a specific communication is tailored not only for the the data structure, the processor structure, and the mapping, but also for the specific data access involved in that communication, that expression can often be much simpler. We currently do optimizations of this type for a few simple cases and have found that the send and receive times do not differ very much from those generated by hand coded programs. In the Intel iPSC2, the start up time per message has increased from 300 μsec to 600 μsec but the transfer time per byte remains the same.

We believe that we need to adopt this strategy for all communications in DINO.

5.3.2 The Message Typing Problem. Because of the semantics of DINO, we assign a message type to each message involved in a send/receive that uniquely identifies the "name" of the data being sent. By "name" we mean not only the variable name assigned to the particular data structure but also the exact range (sub-array) of the particular part of the structure involved in the communication. We then use the message type provided by Intel on the iPSC1 or the iPSC2 to receive only the particular message we are looking for.

However, the possible number of message types for for the data structures in most programs is larger than the size of the type integer provided for either machine. In these cases, we are forced to put a header on the message which contains the type, receive these messages, examine them so see if the type is correct, and store them in buffers if it is not.

For example, in order to cover all possibilities, for a 1024 by 1024 array we must reserve about 2^{40} message types. This exceeds the available message types on both the iPSC1 and iPSC2. However, if the only access to the this array is of the form:

```
A[variable][]
```

then there are only 2^{10} message types needed. Thus, if the compiler examines the whole program and looks at all of the sends and receives for each data structure, it can often significantly reduce the number of message types needed so that we actually stay within the number provided by the machine. We do this particular optimization in the current DINO compiler.

5.4 Potential Future Improvements

Finally, we discuss several improvements that could be made which we believe would result in better performance of programs using our mapping approach. The first of these addresses the `[compress][wrap]` problem described above. The others would simply provide us with additional improvements in certain cases if we could interact with the machine at a lower level.

5.4.1 Transposing Data Structures. In an example that we discussed earlier, we showed how using a `[compress][wrap]` mapping could seriously affect the performance of the program. While some of this problem can be ameliorated by using the techniques described above in Section 5.3.1, some of it is irrevocably tied up with the particular mapping. If a programmer is going to do column wraps, there will be a $O(N^2/P)$ problem whether it is done by hand or in a high level language.

One possible solution to this problem is for the compiler to examine the mapping and the particular data structure, and warn the programmer when the particular combination is likely to result in a significant performance degradation. The programmer could then decide if the particular algorithm allowed the data structure to be transposed or if a different mapping could be used. This solution is possible because high level mapping information is supplied to the compiler.

5.4.2 Interacting with the Operating System. Finally, there are some possible improvements we could make if we had more control over how the underlying communications operated. We mention three possibilities.

First, in an architecture that permitted it, it might be faster to open communication channels at the beginning of a program (or a procedure) and leave them open for many messages. The compiler would have to do enough analysis of the communication patterns to know that it would be possible to get all the messages through if this was done. High level mapping information makes this feasible.

Second, if we could provide the operating system with information about the structure of data which is being communicated and which happens not to be stored as single physically contiguous piece, then the operating system could directly copy to or from that data structure. This would avoid the additional bu"013ering and copying step that we have to do now in this case.

Finally, if the operating system was able to accept much larger message types, we could avoid the duplicate buffering and copying that occurs when we are unable to assign unique message types under the current system. In addition, we could eliminate the current compiler overhead of optimizing the message type generation.

6. Future Research

In addition to implementing some of the changes we discussed above, we intend to explore several additional mapping primitives at some time in the future.

The first is not a new primitive that the programmer uses, but rather an operation that the compiler does. Currently, it is not legal to have a data structure with fewer dimensions than the structure of processors unless the data is mapped "all" (a copy of the whole data structure goes to each processor). We want to redefine the mapping process so that the programmer defined mapping is simply replicated along the axis(es) of the processor structure that have not been aligned with any axis of the data structure. This would allow any combination of data structure dimension and processor structure dimension.

Second, we want to add a "ring" primitive. This would specify that a particular data axis would appear to be a ring for purpose of overlap copies.

Third, we want to add several variations on the communication implications of the "all" primitive. Currently, that primitive defines index zero of the axis to be the "home" copy for communications purposes. Which index is the home processor should be under the control of the programmer.

Fourth, we want to explore mappings for more complex communication patterns. An example would be a power of two offset mapping. This mapping would specify that the two copy processors were a programmer defined power of two away on the data axis and would be particularly useful for fast fourier transform algorithms.

Finally, we want to explore dynamic mappings, that is, mappings where the mapping definition changes during the program execution.

7. Summary

We believe that our use of these structured mappings has provided us with both a number of benefits and a number of problems. There are four primary benefits.

Most importantly, our approach provides the benefit of greatly simplifying how the programmer specifies data decomposition.

Second, it simplifies inter-process communication for the programmer by making it unnecessary to specify the destination or source of sends and receives and by handling parameter distribution and collection automatically.

Third, it provides the programmer with a large number of choices using only a few primitives. We believe that this mapping process is reasonably easy for a programmer to master, because only a few primitives need be used to generate a large number of mappings. In addition, DINO allows the programmer to predeclare mappings and simply reference them by name. This allows us to construct libraries of common mappings.

Finally, it works relatively efficiently for most of the common mappings we have tried.

These benefits do not come without a price. Most obviously, there is some performance degradation in a few cases. How serious these will be is difficult to tell until we can try some of the improvements we propose.

References

[1] D. Callahan and K. Kennedy. "Compiling Programs for Distributed Memory Multiprocessors." *The Journal of Supercomputing*, 2:151-169, 1988.

[2] R. Bjornson. "Experience with Linda on the IPSC/2." *The Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, 493 - 500, March 1989.

[3] P. Mehrotra and J. Van Rosendale. "Compiling High Level Constructs to Distributed Memory Architectures." *ICASE Report*. Number 89-20, March 1989.

[4] M. Rosing, R. Schnabel, R. Weaver. "DINO: Summary and Examples." *Proceedings of the Third*

Conference on Hypercube Concurrent Computers and Applications, 472 - 481, January 1988.

[5] D. Socha. "Spot: A data parallel language for iterative algorithms." Submitted to ICPP90, January 1990.